MiniMat: a flexible minimal language to build matrix expressions

Terence Lim - tl2735@columbia.edu

June 8, 2016

Proposal

I propose a minimal language, MiniMat, to support matrix-based programming, which contains a core set of primitives that can be assembled into more complicated abstractions for matrix expressions, linear algebraic formulas and statistical algorithms.

The compiler front end components (lexer, parser, semantic analysis and intermediate code generation) will be written in OCaml, OCamllex, OCamlyacc and OCaml Llvm, to translate source language into LLVM IR target code.

Usage

MiniMat is aimed at programmers from technical domains who work primarily with matrix expressions. The language will treat the matrix as a primitive data type and accept syntax so that matrices are easy to construct, initialize and subset, in addition to providing imperative statements for loops (e.g. for and while), conditional flow (if-then-else), and scalar arithmetic and boolean operators.

Rather than initially providing a legend of built-in functions and matrix operations, MiniMat will focus on providing a core set of primitives from which users can build up the sort of abstractions they would like. For example, we could use the MiniMat language itself to define the matrix multiplication operation, and bind this to a new infix operator, say "`.*`" (similarly, other new arithmetic and boolean operators for matrices can be defined using the language from the corresponding primitive scalar operators). As another example, some applications may care whether two matrices are *row-equivalent* (i.e. one can be changed to the other by a sequence of elementary row operations), hence we could use the MiniMat language to code up a new boolean equality operator, e.g. `A .~ B` where `A` and `B` are of type `matrix`, that computes and compares their reduced row-echelon forms. After defining such operators, we can build-up further abstractions of matrix expressions and algorithms.

Specific course-oriented goals are addressed below:

- The user can express a variety of algorithms. The MiniMat language includes imperative programming statements such as conditionals (e.g. if-then-else) and loops (e.g. for, while), scalar arithmetic and boolean operators, and matrix data types.

- The language provides primitive building blocks. Although MiniMat targets matrix-based programmers, it will have few built-in matrix functions: rather, the MiniMat language itself will be used to define matrix operators and a suite of support functions.
- The compiler lowers the level of abstraction. It will translate high-level matrix and programming statements into low-level LLVM IR instructions.

Language

In developing MiniMat, we shall endeavour to identify and include only what is absolutely primitive, so that its ultimate intended usage, for matrix expressions and calculations, can be expressed as code in the language itself. Chart 1 describes the primitive parts of the language: in addition to scalar operators, control flow statements and function definitions commonly specified in an imperative language, MiniMat shall offer the `floatmat` (two-dimensional array of floats) and `intvec` (one-dimensional vector of ints) data types, as well as accept syntax for constructing, selecting from and populating a matrix. Chart 2 lists the necessary primitive "helper" functions to be written in the MiniMat language to help the code generation component translate the matrix expression syntax.

Chart 1: Proposed Language Primitives

Category	Symbols	Description
Data types	`int bool float string`	integer, boolean, floating point, character string
	`floatmat`	two-dimensional matrix of floating points
	`intvec`	one-dimensional vector of integers
Scalar operators	`+ - * /`	arithmetic
	`< == > <= >= !=`	comparisons
	`&& || ! true false`	logical
Matrix expressions	`2:5`	vector of int values: 2, 3, 4, 5.
	`[1, 2, 3; 4, 5, 6]`	initialize a matrix with values
	`A(2:4,6)`	select rows 2–4, col 6 of matrix A
	`A(d,2)`	select col 2, rows in vector `d` of ints, of matrix A
	`A'`	transpose
	`rows cols length`	size of matrix or vector
	`matnew vecnew`	constructors
	`matput vecput`	put a value at an index location
	`matget vecget`	get a value from an index location
Control flow	`if` (*expr*) `then else`	if statement
	`while` (*expr*)	while loop
	`for` (*expr*; *expr*; *expr*)	for loop
Built-in functions	`print exception`	print, raise runtime exception
	`defop`	bind function to an operator symbol
	`matextern vecextern`	call external c functions

Chart 2: "Helper" functions to be written in MIMIMAT language

Category	Functions	Description
Construct matrix	`catcol catrow catvec`	help build-up matrix from expression of the form: `[1 2 3; 4 5 6]`
Construct vector	`vecints`	help build-up vector of int values from a *colon* expression of the form: `2:5`
Get matrix values	`matselect vecselect`	help select matrix of values from subset of matrix in expression of the form: `B = A[2:3,d]`
Assign matrix values	`matassign vecassign`	help assign matrix of values into a subset of a matrix in expression of the form: `A[2:3,d] = B`
Matrix operators	`matmul`	implement matrix multiplication and bind to `.*` infix operator

Sample Program

The parts of the language and what they do can be illustrated by the following example. First it defines a boolean function with two matrices as arguments and detects if they are equal, in the sense that every pair of corresponding matrix items is equal; this is bound to the "`.==`" operator. Next it defines a new function that converts a matrix to reduced row-echelon form. Finally, it defines a boolean function to detect if two matrices are row-equivalent, using earlier definitions; this is bound to the "`.~`" operator. The `main()` function labels the program entry point, where two sample matrices are initialized with values and compared for row equivalence.

Listing 1: MiniMat sample code

```
/* detect two equal matrices */
bool mateq(floatmat a, floatmat b) {
  int i;
  int j;
  int m;
  int n;
  m = rows(a);
  if (m != rows(b)) exception("mateq: unequal rows");
  n = cols(a);
  if (n != cols(b)) exception("mateq: unequal columns");
  j = n+1;
  for (i = 1; i <= m && j > n ; i = i + 1) {
    for (j = 1; j <= n && a[i,j] == b[i,j]; j = j + 1) {
    }
  }
  return (i > m && j > n);
}

/* bind binary operator to detect matrix equality */
defop mateq ".==" ;


/* define function to compute reduced row echelon form of a matrix */
matrix rref(floatmat a) {
```

```
25    int i;
26    int j;
27    int k;
28    int end;
29    floatmat tmp;
30
31    end = rows(a);
32    while (i <= end && j <= end) {
33      k = maxindex(abs(a[i:end,j]));
34      k = k + i - 1;
35      tmp = a[k,j:end];
36      a[k,j:end] = a[i,j:end];
37      a[i,j:end] = tmp;
38      tmp = a[i,j:end] ./ a[i,j];
39      a[1:end,j:end] = a[1:end,j:end] - a[1:end] .* tmp;
40      a[i,j:end] = tmp;
41      i = i + 1;
42      j = j + 1;
43    }
44    return a;
45  }
46
47  /* to detect row equivalence; use rref function to compute reduced
48  row echelon form and .== operator to detect matrix equality */
49  bool matroweq(floatmat a, floatmat b) {
50    return (rref(a) .== rref(b));
51  }
52
53  /* bind binary operator to detect matrix row equivalence */
54  defop matroweq ".~";
55
56  void main() {
57    floatmat a;
58    floatmat b;
59    a = [5.0 3.0 4.0; 2.0 2.0 4.0; 1.0 2.0 1.0];
60    b = [3.0 1.0 0.0; 2.0 2.0 4.0; 1.0 2.0 1.0];
61    if (a .~ b) print "Is row-equivalent";
62    else print "Not row-equivalent";
63  }
```